Using Naive Bayes and Bi-directional LSTMs to classify sentiments in Tweets

Final Project Submission for IDS 703, Fall 2021

Sarwari Das, Deekshita Saikia, Aarushi Verma

December 12, 2021

Introduction

The last few decades have witnessed an exponential growth in social media content, such that researching the sentiment of text generated by these channels have wide-ranging applications to fields like market research, predicting political leanings, monitoring reviews or reputations for brands, and even web browsers that detect bias in articles. In many ways, sentiment analysis lies at the intersection of many fields such as Natural Language Processing (NLP), Computational Linguistics, and Data Mining.

Given its applications, we believe that classifying sentiment from social media is especially germane now, since the way people communicate online has evolved over the years to use ambiguities in natural language, slangs, or even indirect sentiments (from emojis for example), which make the problem of classifying text via traditional machine learning approaches (like SVMs) more complex. To this effect, for our final project we decided to investigate the sentiments of user tweets via two approaches: a Naive Bayes classifier and Bidirectional Long-Short Term Memory neural network (BiLSTM).

Twitter is a popular micro-blogging service where users post real-time messages ("tweets") about their opinion on a variety of topics. The content on Twitter ranges from views on current affairs, reviews for products or simply complaints about daily life. These tweets have become a source to gauge people's sentiments on a massive variety of topics. However, sentiments on Twitter can be expressed implictly or explicitly; we believe that this can present a challenge undertaking due to their complexity, ambiguity, the morphological richness of their language and the absence of contextual information. In this report, we will compare the performance of a probabilistic model with a neural network, both of which classify tweets into positive or negative sentiments based on their content. Our analysis can be found in the Github repo here.

Data

The dataset we use for the purpose of this analysis is Sentiment140, sourced from help.sentiment140.com. The data set was created by graduate students at Stanford University to analyze sentiments based on users' tweets about sentiments on a variety of brands, products and topics.

There are 6 fields in the original dataset, with emoticons removed:

- Polarity of the tweet (0:Negative, 2:Neutral, 4:Positive)
- The id of the tweet

- The date of the tweet
- The query. If there is no query, then this value is NO_QUERY.
- The username that posted the tweet
- The text of the tweet

The tweets in this dataset are obtained from the Twitter API with specific queries, which are arbitrarily chosen from different domains. For more information on the creation of this dataset, please refer to the paper here.

Data Pre-Processing

The original dataset contains 1,600,000 tweets. For our analysis, we proceeded with a sample of this data. Our filtered dataset contains 40,000 tweets, wherein we modify the labels of the tweets from 0:Negative, 4:Positive to 0:Negative, 1:Positive. Our sample dataset has an equal number of tweets tagged to each sentiment.

To clean the tweet texts for analysis, we leverage the `regex` and `nltk` libraries. In order to correctly classify sentiments based on the text, we removed any usernames, email ids and URLs present in the tweets. Further, we also removed punctuation marks, numbers, special characters and multiple spaces from the text since they do not aid in determining the sentiment of a tweet.

We also removed stopwords using the `nltk` stopwords dictionary. Stopwords mainly include common words such as 'a', 'the', 'in' which take up processing time but may not provide any useful information for the purpose of sentiment analysis. We also carry out stemming and lemmatization to return only the root word (stem/ lemme respectively).

Modeling

Using the Sentiment140 dataset, we train a Naiye Bayes classifier, as well as a BiLSTM to classifiy the sentiment labels based on the text of the tweets. The modeling approach is discussed in detail below.

Generative Probabilistic Model

We choose a multinomial Naive Bayes model as our generative probabilistic classifier. Let us assume random variables Y and $X_1 \dots X_d$, corresponding to class y and the observables $x_1 \dots x_d$ respectively. Naive Bayes models the joint probability

$$P(Y = y, X_1 = x_1, \dots X_d = x_d)$$

for any class y associated with the attributes $x_1, \dots x_d$. The model "naively" assumes that the attributes are independent of each other. This reduces the problem to learning two "parameters": the prior probabilities of different classes, and the likelihood of different features for each class for the underlying sample. This is denoted by

$$P(Y = y) \times P(X_1 = x_1, \dots, X_d = x_d \mid Y = y)$$

The values for these parameters are estimated with Maximum Likelihood Estimates (MLE).

A generative model includes the distribution of the data itself, which enables us to see how likely a given example is. It produces convincing target classes by generating attributes that fall close to their real counterparts in the underlying data. When used for text classification, Naive Bayes is an example of a bag-of-words model, since it does not take into account the order of the words.

With the pre-processed dataset, we proceed to fit a Naive Bayes classifier with `scikit-learn`. A 75-25 split of the dataset is created, the former to train the classifier on, and the latter, to test the performance of the classifier. We create a pipeline that executes a Term Frequency-Inverse Document Frequency (TF-IDF) transform, followed by a Multinomial Naive Bayes fit. We also construct a grid of hyper-parameters to tune the pipeline over, which are described below:

- `n_gram_range`: n-grams to train the model on. We include both unigrams and bi-grams, to capture sentiments like *"not bad"* and *"very good"*.
- `use_idf`: Use inverse document frequency re-weighting.
- `smooth_idf`: Smooth IDF weights by adding one to document frequencies, to prevent zero divisions.
- `norm`: If the absolute values of the vector elements sum to 1 ("l1"), or the squares of the vector elements sum to 1 ("l2"), so that each output row has unit norm.
- `max_features`: The maximum length of the vocabulary to train on.
- `alpha`: Lidstone/Laplace smoothing parameter, to account for words not presented in the training sample. We consider the values $[0.1, 0.5, 1, 1.5, 2]$.

A grid search is performed to choose the hyperparameters that yield the least loss in the test sample. We obtain the following optimised parameters post grid search:

- `n_gram_range`: (1,1)
- `use_idf`: False
- `smooth_idf`: True
- `norm`: l1
- `max_features`: 25000
- `alpha`: 0.5

The classifier is then fit with the tuned parameters, except for `n_gram_range`, which we retain as (1,2) to train on both unigrams and bi-grams.

Discriminative Model

Our initial approach to designing a discriminative neural network for sentiment analysis was through a recurrent neural network (RNN). Here, we are cognizant of the fact that RNNs suffer from a vanishing gradient problem when handling long sequences of data. LSTMs can circumvent this problem, at the cost of failing to take some contextual information into consideration. However, we believe that extracting a sentiment highly depends on the tweet's contextual information. Therefore, the sentiment analysis approach in this report uses a Bidirectional LSTM Network (BiLSTM) which has the ability to extracting the contextual information from the feature sequences by dealing with both forward and backward dependencies.

We iterated through several models before finalizing on a discriminative neural network to address our objective of tagging tweets to their associated sentiment. Our initial model preprocessed the data using Natural Language Toolkit (NLTK) and employed an LSTM to tag tweets to sentiments. In the next iteration, we used the same neural network architecture but used pre-trained Word Embeddings (GloVe) to process our data. We believed that our results could be improved by adjusting the network, so our final model was a bi-directional LSTM with GloVe embeddings.
Before implementing our network, we had to take several steps to ensure that our inputs were in their appropriate forms. We took the following measures before accepting inputs into our neural network:

- **Tokenization**: In our network, we used the Spacy library to tokenize our text. The Spacy tokenizer splits the text on whitespace similar to the split() function, then checks whether the substring matches the tokenizer exception rules. For example, "don't" does not contain whitespace, but should be split into two tokens, "do" and "n't", while "U.K." should always remain one token. Next, it checks for a prefix, suffix, or infix in a substring, these include commas, periods, hyphens, or quotes. If it matches, the substring is split into two tokens. After tokenization, we convert our data into PyTorch's TabularDataset.

- **Embedding**: Neural networks require that we pass in integers as our input laters. We use GloVe (Global Vectors for Word Representation), which is an alternate method to word2vec for efficiently learning word vectors, and to create word embeddings. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. Here, the maximum vocabulary size we've chosen is 25,000 words, even though 76,000 unique words were present in our data. Given that tweets often have invalid words, we believed this limit was justified. GloVe embeddings were trained on 6 billion tokens and the embeddings are 100-dimensional.

- **Sequencing and Padding with BucketIterator**:

  This iterator rearranges our data so that similar lengths of sequences fall in one batch with descending order to sequence length. Further, BiLSTM requires that length of all sentences have to be the same length. So, the iterator pads each tweet to be the same length to process in batch. Overall, for example, "I like the U.K" can be tokenized to ["I", "like", "the", "U.K]. Converting the tokenized sentences into sequences would look like [1, 2, 3, 4]. Then padding the sequences would look like [1, 2, 3, 4, 0, 0]. The two zeros after the number 4 are the padded sequences. The BucketIterator will group tweets of similar lengths together for minimized padding in each batch.

Model Architecture

A diagram representing the structure of our neural network is given below. The model contains three essential subparts: the embedding layer, the bidirectional LSTM layers, and the final densely connected layers. This model is implemented and trained PyTorch. We use the training data to optimize model parameters using backpropagation while comparing the loss and accuracy on the validation model to optimize the hyperparameters. We experiment with changing learning rates, batch sizes, the maximum vocabulary size, number of epochs to run, and the different layers in our neural net. We apply binary cross entropy loss as the loss function to the final Linear layer in the model to generate our final topic classifications.
Our hyper paramters are:

- Input size: The number of features for each element in the input in our model. In our case, the input dimensions is equal to the dim of pre-trained GloVe vectors (100).

- Number of Layers: In our case, we set this argument to 2 which means that the input x at time t of the second layer is the hidden state h at time t of the previous layer multiplied by dropout.

- Batch size: The model takes 256 in each iteration to train them.

- Dropout: If this argument is greater than zero, it will produce a layer with dropout probability on each output of the LSTM layer except the last one. Our Dropout is set at 0.5.

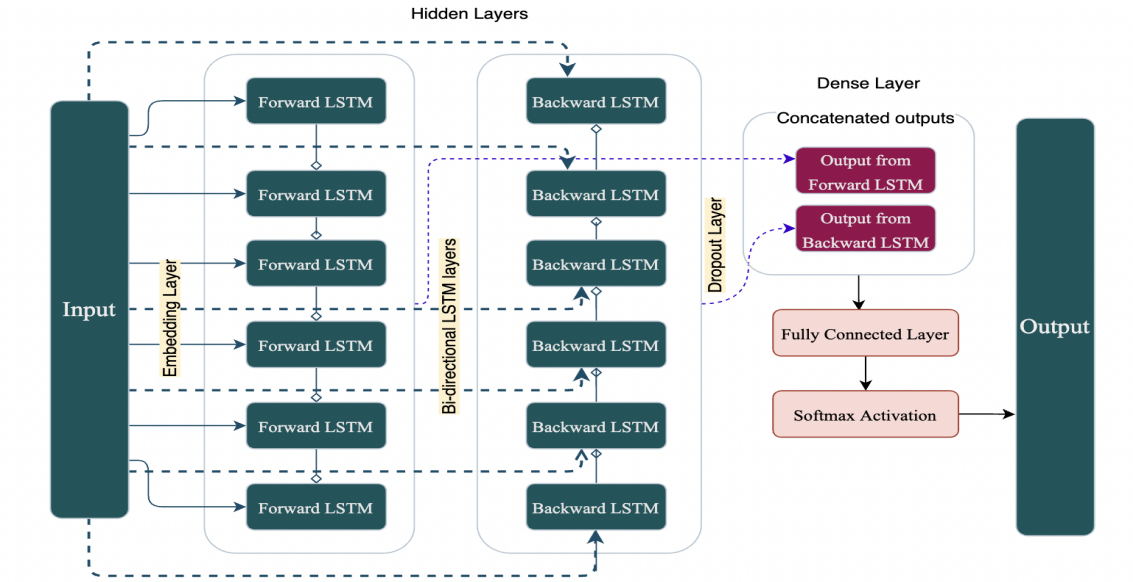
- Bidirectional: By changing bidirectional variable modes we can control the model type. Ours is True since it is a bidirectional LSTM.

- Optimizer: We're using a Adam Optimizer with Learning Rate set at 0.002.

Figure 1: Model Summary

```
LSTM(
  (embedding): Embedding(25002, 100, padding_idx=1)
  (encoder): LSTM(100, 256, num_layers=2, dropout=0.5, bidirectional=True
  (predictor): Linear(in_features=512, out_features=1, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

Figure 2: Neural Network architecture



To summarize, our model accept tweets of varying lengths, which are fed into a embedding layer with dimensions = 100 (after being padded). Next, inputs are passed to a hidden layer; a bi-LSTM neural network with 2 layers with a batch size of 256. Although the final layer is the output layer with a linear activation function, the final outputs are provided using a sigmoid function to round predictions to the closest integer since our problem is a binary classification one. We have also used Dropout layers to prevent overfitting.

Analysis

We apply both models to synthetic data generated by Naive Bayes and real data on Tweets.

Generating synthetic data with Naive Bayes

After a Naive Bayes Classifier is fit to the dataset, an arbitrary array of random classes is generated. Then, corresponding to each class in this array, words are sampled from a multinomial distribution, using the probabilities of features from the Naive Bayes classifier. Since we observe that the average length of all the tweets in our dataset amount to roughly 7 words, we randomly choose 7 unigrams and bi-grams from this distribution and concatenate them to generate a synthetic tweet. The data generated looks like as shown in figure 3 below.

	label	text
0	1	seven hours one get julyyyyy exitedd breast la...
1	1	higher mine christopher ugghhh fishing otherwi...
2	1	havegood night dock meetings scores slid canwa...
3	1	notts hadheadache days today hope whoop whoop ...
4	0	test thank wheres herecome sometime soon world...

Figure 3: Synthetic data generated with Naive Bayes classifier

Applying Naive Bayes to real data

With the optimal parameters obtained via grid search, the Multinomial Naive Bayes classifier is now fit on the train sample and tested on the validation sample. The classifier yields an accuracy of 83% on the train dataset and 73% on the test dataset. The classificaton metrics are as shown below:

	precision	recall	f1-score	support
0	0.7449	0.7133	0.7287	4994
1	0.7233	0.7542	0.7384	4963
accuracy			0.7337	9957
macro avg	0.7341	0.7337	0.7336	9957
weighted avg	0.7341	0.7337	0.7336	9957

Figure 4: Classification metrics of Naive Bayes classifier on Sentiment140

Applying Naive Bayes to synthetic data

The Naive Bayes classifier trained on the raw tweets is applied on the synthetic data generated from the previous step. This yields an accuracy of 100%. This is expected as the synthetic data was generated using the very same classifier. The classification metrics are as shown below:

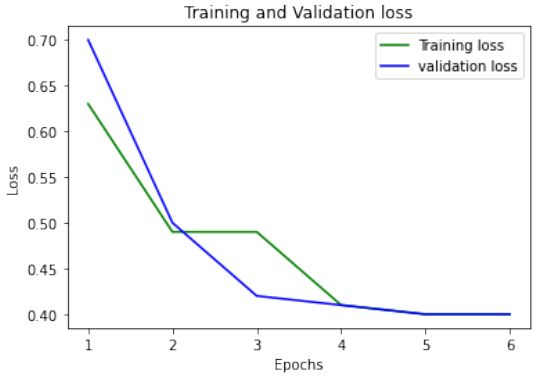
	precision	recall	f1-score	support
0	1.0000	1.0000	1.0000	1
accuracy			1.0000	1
macro avg	1.0000	1.0000	1.0000	1
weighted avg	1.0000	1.0000	1.0000	1

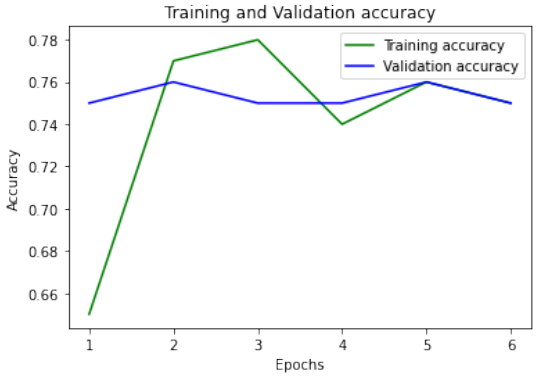
Figure 5: Performance of Naive Bayes classifier on Synthetic Data

Applying bi-LSTM to real data

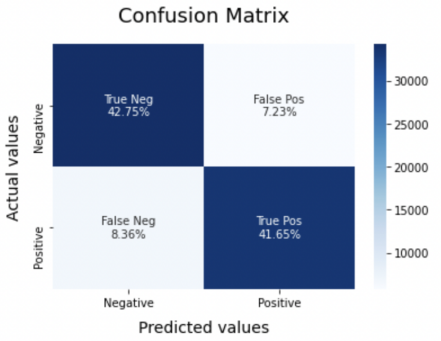
The real tweet data set is split such that 10 percent of the entire data is assigned as a test set, 10 percent is the validation set, and the rest remains as the training set. The training data set is fed to the bi-LSTM with a batch size of 256, and is run through 6 epochs. The network is using an Adam Optimizer with a learning rate of 0.002. The loss and accuracy data of the model for each epoch is shown below. At the end of 6 epochs, the model has a validation accuracy of 74.25 percent. Given that loss and accuracy values are close and converging, we don't think that our model suffers from an overfitting or underfitting problem.

Below, we can see how the model predicts sentiment for our test data. We believe that our model fails for tweets that are too short or too generic, such that many words in them are omitted due to being stop words. Further, misspelt tweets are most likely ignored after the cleaning process, and so are often tagged correctly. The same applies to non-english tweets as well. From the confusion matrix, it can be concluded that the model makes more False Negative predictions than positive. This means that the model is somewhat biased towards predicting negative sentiment.



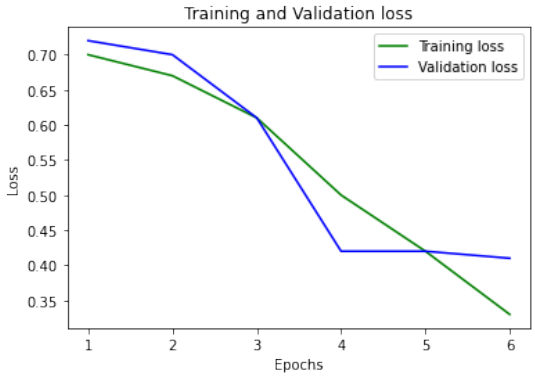


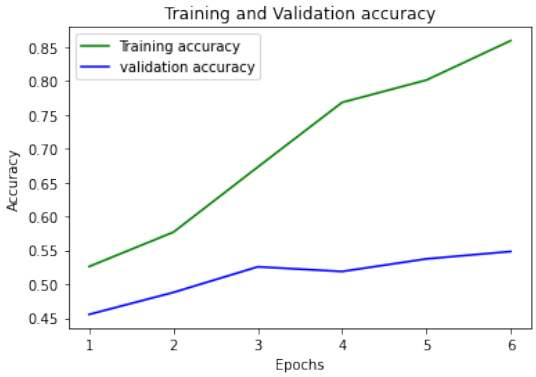
	Tweet	Prediction	True Label
0	is singing the blues, waiting for the last wee...	0.076035	0
1	goin 4 a bit of shoppin, den continue revisin ...	0.556183	0
2	@punkygumboot haha, to be honest i do agree, b...	0.456077	1
3	recording bass tracks . multiple punch - in / ...	0.823227	0
4	i had my galaxy .. and god it was good!! whe...	0.178978	0
5	i love my best friend, she is amazing in every...	0.880249	1
6	well mer iz being big b!% (* but d' mall wuz s...	0.375285	1
7	ya face is	0.783655	1
8	@my_lyrical_lies wow! njo, i haven't bin yet...	0.272490	0
9	@nigs was going to chilling but get back to wo...	0.593274	0



Applying bi-LSTM to synthetic data

Next, we test the synthetic data generated by Naive Bayes against our bi-LSTM neural network. The graphs below show the loss and accuracy of the model. Although both graphs are as expected, i.e, loss decreases with increasing epochs while accuracy increases, the overall accuracy of our model (for test data) is quite low at a 53.67 percent.





Conclusion

We realize there are several problems with trying to classify sentiment of Tweets through our approaches, many of which have to do with the underlying data. For example, Twitter witnesses a widespread use of acronyms which we did not account for. Our accuracy rates could be affected by this as acronyms they would make no contribution to our model. Further, approaches are not always able to accurately classify those reviews where the text contains sarcasm, irony or humor. In the images below for example, our model predicts positive sentiment for both images, although they reflect negative sentiment. The same applies for slangs, non-english terms, and tweets heavily embedded in context. Further, we must recognize that although we fulfilled all required steps for data-cleaning, the process is not perfect. For example, to preserve the meaning of the word (school vs schol for example), we skipped over removing repeated characters in the text. However, this fails to clean words like "heyyyyy", which creates noise in our data. Next, although we did remove single letter words, our tokenizer works such that "hasn't" for example, gets tokenized to 'hasn' and "t', which recreates the problem in our tokenized data. This further leads to a sparser vocabulary matrix.

In general, we believe this model is a relatively good fit given that neither of our approaches were computationally expensive. At any given time, the Naive Bayes Classifier did not require more than 15 GB of ram, while the maximum time an epoch in the Bi-LSTM took was approximately 18 minutes. We recognize that while this type of biLSTM is able of capturing contextual information and long-term dependencies, we can experiment with hybrid models containing CNN and bi-LSTM to capture both the local features of the texts and their global and temporal semantics. We can also explore the usage of BERT as a language model to represent the tweets. The main advantage of BERT compared to other models is its ability to generate contextualized word embeddings. These could all be ideas for future extensions of this project.